Data manipulation with Rcell (Version 1.1-6)

Alan Bush

November 3, 2011

1 Introduction

Once you have your data loaded into **R**, you can filter it and plot it as shown in "Getting Started with Rcell". To read that document type in the console

```
> vignette("Rcell")
```

But many times we want to do some manipulation or transformations on the data before plotting it. In this document you'll see how this can be done using **Rcell**.

2 Transforming variables

If you haven't done so, load the **Rcell** package and the example dataset with

```
> library(Rcell)
> data(ACL394filtered)
```

The easiest way to modify your dataset is to create new variables from existing ones. For example, its desirable to correct the fluorescence measure of a cell by the background fluorescence. To do this for the YFP channel we can use the `f.bg.y` variable, that contains the most common value (mode) for pixels not associated with any cell. If a cell has no fluorophores, we expect it to have a total fluorescence equivalent to `f.bg.y` times the number of pixels of the cell, `a.tot`. So the background corrected fluorescence can be calculated as `f.tot.y - f.bg.y*a.tot`. To creare a new variable called `f.total.y` with the corrected value for fluorescence we can use the `transform` funcion. As all other **Rcell** functions, the first argument is the `cell.data` object to tranform.

```
> X <- transform(X, f.total.y = f.tot.y - f.bg.y * a.tot)
```

Once created, you can use the new variable as any other variable of the dataset. You can create several variables in a single call to `transform`, as shown next for the fluorescence density variables.

```
> X <- transform(X, f.density.y = f.tot.y/a.tot, f.density.c = f.tot.c/a.tot)
```

You can keep track of the variables you've created with the `summary` function, that will display among other things the "transformed" variables with their definition.

```
> summary(X)
```

pos	alpha.factor
1	1.25
2	1.25
3	1.25
8	2.50
9	2.50
10	2.50
15	5.00
16	5.00
17	5.00
22	10.00
23	10.00
24	10.00
29	20.00
30	20.00
31	20.00

Table 1: example data.frame to merge

3 Merging variables

Sometimes there is no formula to specify the new variable you want to create. For example, you might want to create a variable that describes the treatment each position received. In the example dataset (`help(ACL394)`) each position received a different dose of alpha-factor pheromone, according to the Table 1.

You can create this table in Excel[1] and save it as a tab delimited text file. If you name it "mytable.txt", then you can loaded into **R** with `read.table`. The best option is to save the file in your working directory, or to change your working directory to where you saved the file (see `?setwd`).

```
> mytable <- read.table("mytable.txt", head = TRUE)
```

If the first row of your text file contains the column names (recommended), you have to set *head* to `TRUE` in `read.table`. Once loaded you can add the new data to your dataset using the `merge` function. This function looks for common variables between `X` and `mytable` and, if it finds them it merges the dataset according to those common variables. Be aware that the names of the columns of `mytable` have to match EXACTLY[2] to the variables of `X`[3]. In this case it will merge by *pos*. You can also specify the variable to merge by with the *by* argument.

```
> X <- merge(X, mytable)
```

```
merging by pos
merged vars:
  alpha.factor: numeric w/values 1.25, 2.5, 5, 10, 20
```

4 Transform By

A common transformation is normalization, i.e. dividing the value of a variable by the "basal" level. For example, we might be interested in the fold icrease of YFP fluorescence through time. So we need to divide the measured value at each time by the value at time cero, and we need to this for every cell. How can we do this? The steps we should follow are the following:

[1]or from R: `mytable<-data.frame(pos=with(X,unique(pos)),alpha.factor=rep(c(1.25,2.5,5,10,20),each=3))`

[2]**R** is case-sensitive so `"pos"` is different to `"Pos"`

[3]You can see these variables with `summary(X)`

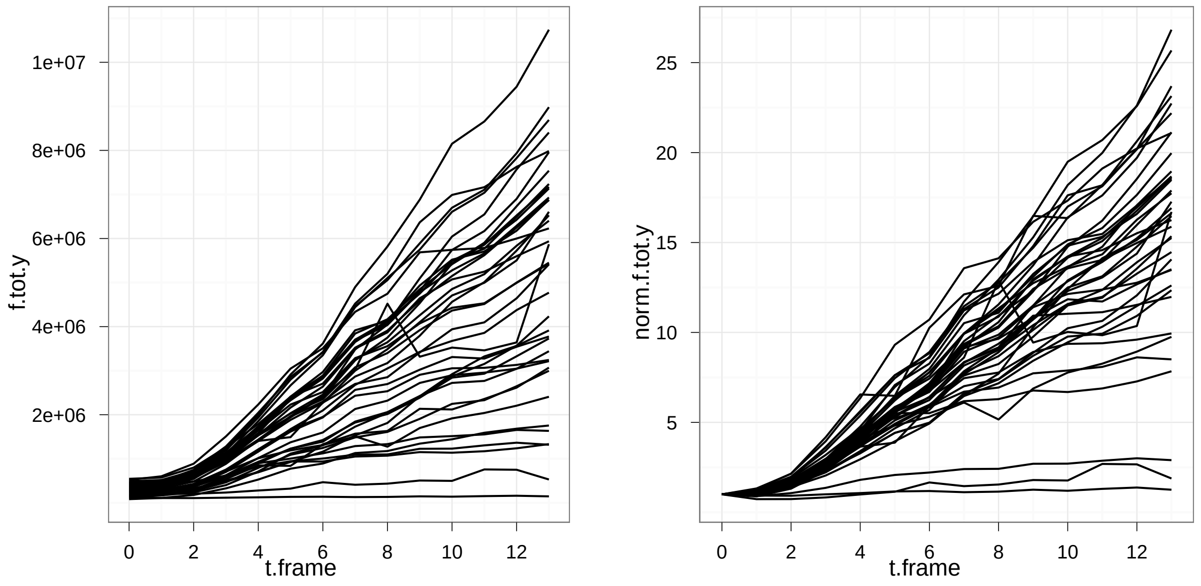

Figure 1: Left: raw single cell time course for YFP fluorescence. Right: Same data normalized to each cells value at time cero.

1. Divide the dataset by cell, creating a table for each cell.
2. Indentify the value of fluorescence for time cero.
3. Create a new variable by dividing the fluorescence at each time by the value at time cero.
4. Join the cells datasets back together to retrieve the original dataset with the new variable.

All these steps are done by the function `transform.by`, but it requires information on how each step should be done. For the first step, it needs to know how to partition the dataset. This is specified by passing a quoted list of variable, whos combination of levels specify a group. For example, if you want to divide the datset by position, the second argument of `transform.by` should be `.(pos)`. If you want to divide your dataset by cell use `.(pos,cellID)`. Note that cells in different position can have the same cellID, so the combination of `pos` and `cellID` uniquely identifies a cell. The variable `ucid` (for Unique Cell ID) is another way to uniquely identify a cell. Next we need to specify the name of the new variable to be created (`norm.f.tot.y` for example), and the definition for this variable, `f.tot.y/f.tot.y[t.frame==0]`. With the square brackets we are selecting the value of `f.tot.y` when `t.frame` is cero. Remember to use the logical operator `==` and not the assignation operator `=` within the brackets.

```
> X <- transform.by(X, .(pos, cellID), norm.f.tot.y = f.tot.y/f.tot.y[t.frame ==
+     0])
```

You can see the raw and normalized data in Figure 1. Another way to normalize the data, is dividing by the mean of the first three values.

```
> X <- transform.by(X, .(pos, cellID), norm2.f.tot.y = f.tot.y/mean(f.tot.y[t.frame <=
+     2]))
```

5 Aggregating your data

To calculate summary statistics you can use the `aggregate` function, that returns an aggregated table. That means that the value of each cell of this aggregated table is caculated from more than one cell of the original table. For example you might be intereset in getting the mean YFP fluroescence for each pheromone dose. `aggregate` accepts two notations that give equivalent results.

```
> aggregate(X, .(alpha.factor), select = "f.total.y")
> aggregate(X, f.total.y ~ alpha.factor)
```

```
  AF.nM f.total.y
1  1.25   1071898
2  2.50   1622198
3  5.00   2234246
4 10.00   2393427
5 20.00   2377602
```

You can calculate other statistics using the *FUN* argument, and you can include more than one variable. Here we calculate the median for `f.tot.y`, `f.tot.c` and `a.tot`. Note the use of the wildcard in the *select* argument.

```
> aggregate(X, .(alpha.factor), select = c("f.tot.*", "a.tot"), FUN = median)
```

```
  alpha.factor   f.tot.c f.tot.y a.tot
1         1.25 1047808.5 1212391 415.5
2         2.50 1055751.0 1564543 415.0
3         5.00 1037465.0 2032817 407.0
4        10.00 1001638.5 2224172 398.0
5        20.00  961167.5 2083660 380.0
```

The partition of the dataset can be done by more than one variable, for example by dose and time. Using the function `funstofun` from the **reshape** package, you can calculate more than one statistic at once.

```
> aggregate(X, f.density.y ~ t.frame + alpha.factor, FUN = funstofun(median, sd),
+     subset = t.frame%%3 == 0)
```

```
   t.frame alpha.factor f.density.y.median f.density.y.sd
1        0         1.25          1037.6060       162.4781
2        3         1.25          2103.6828       420.8901
3        6         1.25          3646.6715       892.4902
4        9         1.25          3916.8426      1112.6042
5       12         1.25          4081.6656      1295.7236
6        0         2.50          1043.6030       141.9683
7        3         2.50          2283.6054       577.7679
8        6         2.50          4663.5886      1387.7653
9        9         2.50          5809.6937      1895.5383
10      12         2.50          6715.9285      2200.4825
11       0         5.00          1057.4033       163.5038
12       3         5.00          2519.7817       662.3112
13       6         5.00          5619.7677      1633.4406
14       9         5.00          8206.8911      2418.8710
15      12         5.00          9600.1285      2859.8918
16       0        10.00          1009.4154       131.5840
```

```
17        3         10.00            2686.9785        508.3823
18        6         10.00            5883.8167       1218.5057
19        9         10.00            9131.4531       2018.9390
20       12         10.00           10792.7466       2478.7091
21        0         20.00            1022.5990        120.6829
22        3         20.00            2730.6114        571.0274
23        6         20.00            6440.7103       1645.1310
24        9         20.00            9512.2292       2497.7941
25       12         20.00           11746.9381       3383.9809
```

6 Evaluating expressions in your dataset

Using the `with` function, you can evaluate a expression in a enviroment created from your dataset. That means that you can use the names of your variables directly, without any prefix. For example to calculate the mean of `f.tot.y` from position 1

```
> with(X, mean(f.tot.y[pos == 1]))
```

```
[1] 1372297
```

If you don't use `with` you have to write the full identifier of the variable, and the code becomes longer and harder to understand. For exaple, the same result can be obtained with

```
> mean(X$data$f.tot.y[X$data$pos == 1])
```

7 Exporting your data

Although you can do much of your analysis using Rcell functions, you might need to export the data to some other application or use another package within **R**. To retrieve the entire dataset in a data.frame, use the double square brackets notation. This returns the registers that pass the QC.filter.

```
> df <- X[[]]
```

This dataset is usually big, and has many varaibles or registers you are not interested in. You can subset the datset as you would a data.frame (but using double brackets)

```
> df <- X[[pos == 1, c("cellID", "f.tot.y", "a.tot")]]
```

You can then save the data.frame to a file with `write.table`, or use it in another **R** package.

For some kinds of data analysis you need your data in a different form than the one **Rcell** uses. You can use the `reshape` function to reshape your data. For instance, a common restructuring is to display time as different columns, and individual cells as different rows. You can obtain this sort of data.frames with the following command.

```
> reshape(X, pos + cellID ~ variable + t.frame, select = "f.tot.y", subset = pos <=
+     2 & cellID <= 10 & t.frame%%2 == 0)
```

```
   pos cellID f.tot.y_0 f.tot.y_2 f.tot.y_4 f.tot.y_6 f.tot.y_8 f.tot.y_10 f.tot.y_12
1    1      1    378752    748712   1350707   2028179   2155404    2072739    2214004
2    1      2    176429    300842    448582    535334    549019     562208     512430
3    1      3    384393    665472   1234888   1913377   2036718    2217148    2071306
4    1      4    245876    510412    887509   1493615   1692185    1987466    2137951
5    1      6    347597    629056   1000791   1533244   1788453    2188437    2381668
```

```
6    1     7   325715    558893    998972   1740997   2080686    2575011    2845928
7    1     8   276242    481790    842095   1310683   1464268    1735160    1826068
8    1    10   314574    559742   1050029   1832519   2166170    2765528    2981627
9    2     2   387551    620656   1049458   1327046   1317000    1409672    1742833
10   2     3   428014    655421   1239405   1616264   1942105    2268256    2583064
11   2     4   452047    718126   1381880   1808801   2260311    2624726    3127893
12   2     5   330852    228660    285463    294746    357041     354951     308372
13   2     6   166711    139176    173661    184098    201449     202782     223541
14   2     7   453124    665657   1288083   1847586   2532047    2676930    2793809
15   2     8   137043    232664    448551    652693    830410     957173    1094667
16   2     9   504990    632012    961822   1340065   1640487    1769006    1929912
17   2    10   317594    440622    763946   1221629   1711411    1788251    1834715
```

see `help(reshape.cell.data)` for more details.

References

Pau, Fuchs et al. (2010). EBImage: an R package for image processing with applications to cellular phenotypes. *Bioinformatics*, 26(7):979-981.

Colman-Lerner, Gordon et al. (2005). Regulated cell-to-cell variation in a cell-fate decision system. *Nature*, 437(7059):699-706.

Chernomoretz, Bush et al. (2008). Using Cell-ID 1.4 with R for Microscope-Based Cytometry. *Curr Protoc Mol Biol.*, Chapter 14:Unit 14.18.